Worksheet 1

An echo of C++

Marking scheme

- Task 1 - 20%
- Task 2 - 20%
- Task 3 - 20%
- Task 4 - 40%

All code should be submitted via a gitlab.uwe.ac.uk repo.

Each task should be documented in a README.md in the main repo, including screen shots of the code running, and details of how your implementation works. Additionally, the code should be structured using header files, along with comments throughout the code.

After completing this worksheet you should be familiar with:

- Defining new dynamic classes
- C++'s use of pointers
- Be able to compile C++11 programs
- Be able to use C+11's smart pointers
- Basic template usage

It is assumed that you are already confident in basic C/C++ programming and this is really just a re-cap. We will continue this theme in worksheet 1, which will look deeper at pointers, which it is really important that you understand for the rest of the module.

The deadline for completing this worksheet is 5th November, 2024, by 2pm.

> IMPORANT: All code must be submitted via gitlab, failure to provide a link to Gitlap repo will result in a mark of 0. No email or other form of submission is acceptable.

Compiling C++ with clang++

Before we begin it is important to be able to compile C++ programs, which you should already be confident at, but in case you are feeling rusty we compile a simple hello world program to get started. Assuming you are logged into the remote server, create a directory to work in and create the a C++ file (hello-world.cpp) with the following content:

```
#include <iostream>

int main(void) {

    std::cout << "Hello World" << std::endl;

    return 0;
}
```

This can be compiled with the following:

```
clang++ -o hello hello-world.cpp
```

and will result in the executable `hello`.

To add support for C++17 features, not used here but relied on later, simply add the command line option `-std=c++17`.

Referencing counting

Languages like C/C++ provide manual memory management, meaning that the programmer must explictly manage the allocation and deallocation of memory with either *malloc/free* or *new/delete*. We will discuss these and other forms of memory management through out the module, including looking at the benifits and downsides, but for now it is worth noting that it is often difficult to manually track the life of an object. Programming languages such as Python and Javascript, along with many others, provide automatic lifetime management. Once everything is done using an object it cam be deleted. Thre are many approaches to this automatic management of memory, covered later in the course, but for now we will look at one of the most simple forms called Reference counting.

Reference counting is a method for automatic memory management that keeps a counter with each object that is created. The counter is the number of references that exist to the object, in the C/C++ case this would be how many pointers refer to this object. Anytime a pointer is copied we increment the count, and anytime a pointer goes out of scope, or is reset, we decrement the count. When the count hits zero the object is deleted since nothing more is using it.

Task 1

To explore this idea of referencing counting we are going to implement a simplifed version of C++'s string class. The class has the following interface:

```
#include <iostream>
#include <cstring>

using namespace std;

class my_string {
public:

  my_string();
  my_string(const char*);

  my_string(my_string const& s);
  my_string& operator= (my_string const& s);
 ~my_string();

  char getChar(const int& i) const;
  void setChar(const int& i, const char& c);
  void print() const;
}
```

If you are unfamilar with copy constructors or overloading the assigment operator, then use Google to find examples and understand what they do.

> You implementation should work with any length of string, e.g. it should use dynamic memory allocation to make a copy of any string data passed in on construction or via assignment. The assignment and copy operators should not make a copy of the string, i.e. they should share the same backing memory, updating one string witll update the other and so on.

> Additionally, for now leave the destructor `~my_string()` empty.

Implement this class in the header `my_string.hpp``. In the filetest_string.cpp`“ add the following code:

```
#include "my_string.hpp"

int main() {
    my_string s("Hello world");
    s.print();
    {
        my_string t = s;
        s.print();
        t.print();
        str::cout << s.getChar(1) << std::endl;
        s.print();
        t.print();
    }
    s.setChar(1,'E');
    s.print();
}
```

Compiling and running this program should produce the following output:

```
Hello world
Hello world
Hello world
e
Hello world
Hello world
HEllo world
```

Task 2

Extend you string implementation to support automatic reference counting. A key element of this is that the interface for `my_string` should not change. However, the destructor `~my_string()` should now free the allocated memory, if and only if, the reference count is 0, i.e. there are no remaining references to the object.

To begin with you can test your referencing counting code with the same code as above, but now the output should be:

```
Hello world [1]
Hello world [2]
Hello world [2]
e
Hello world [2]
Hello world [2]
HEllo world [1]
```

Note that the resulting out now includes the reference count itself, and how it moves to [2] when the string `t` is in scope, and back to [1] when it is goes out of scope.

Task 3

Extend the example test program to demostrate when the case of a reference count of 0.

Task 4

Up until this point the referencing counting has been implemented directly within the string class itself, but what if we wanted to add referencing counting to another class? Clearly we could just implement referencing counting directly in the new class, and again we could do this for the next class we wanted to reference count, and so on. Instead it would make more sense to use a template class that handles referencing counting and is a container for any class that the developer wants to reference count.

For this task you should design a template class that handles reference counting. Rework your string class and examples to use the new referencing counting template class.

> To begin with it would make sense to use a simple class, e.g. point, as a test for your referencing counting class. Once this is debugged and working, only then move on to reworking your string class implementation.

> Make sure that your solutions for the previous tasks are not effected by the changes for task 4.